UNIVERSITY OF SALERNO

Computer Science Department

Master Degree - Software Engineering & IT Management

SOFTWARE DEPENDABILITY PROJECT REPORT

Apache commons-net library analysis

SUPERVISOR

Prof. Dario Di Nucci

AUTHORS

Fabiano Daniele

ID Number: 0522501738

Puca Francesco Maria

ID Number: 0522501797

2023-2024 Academic Year

Contents

CHAPTER 1

Introduction

1.1 Structure of the report

This report contains an analysis of the Apache Commons Net library in regards to its dependability attributes. It is divided into the following chapters:

- Chapter 1 - Introduction: Describes what the report contains, what type of library is analyzed and the evaluation criteria.
- Chapter 2 - Software Analytics: Outlines the CI/CD pipeline's configurations and the examined builds, as well as the results obtained from the SonarCloud analysis, which consist of numerous bugs and code smells.
- Chapter 3 - Bug fixing and Refactoring: Lists all the bugs that were fixed, as well as the methods used to do so. Refactoring efforts related to code smells are also outlined in this chapter.
- Chapter 4 - Testing: Describes the results of the execution of the project's existing test suites. New test cases were generated automatically in order raise their coverage and strength, and Performance testing was conducted, which highlighted methods whose execution times were slowest.

- Chapter 5 - Software vulnerabilities: Security vulnerabilities found through SonarCloud analysis were considered and resolved. Other tools for detection have also been executed, in order to find vulnerabilities in the project's source code and dependencies.
- Chapter 6 - Conclusions: The final chapter of this report contains a table, which sums up the results obtained throughout the previous chapters.

1.2 What is Commons-net ?

Apache Commons Net is a library that implements the client side of many basic Internet protocols, such as Echo, Finger, FTP, NNTP, NTP, POP3(S), SMTP(S), Telnet and Whois.
Originally, this project was a commercial Java library called NetComponents, developed by ORO, Inc. in the early days of Java. The source code was donated to the Apache Software Foundation in 1998 and made available under the Apache License, currently v2.
Commons Net is now part of the Apache Commons Proper project, which has the stated goal of creating and maintaining reusable Java libraries with minimal dependencies on other libraries.
Documentation for the library is provided both as a Javadoc document and on the Apache Commons website[1].
Developers work on Commons Net using its GitHub repository[2]. The JIRA platform is also used as a bug tracker: a JIRA Ticket must be submitted before working on a new issue.
Other open source developers contribute to the project mainly through Pull Requests on GitHub, and communicate with each other using the Commons Developer mailing list.
As of 19/11/2023, the repository for the project has been active since March 2002, and has had 33 contributors help maintain it over the years, with 191 recorded pull requests. 177 forks have been made, and the main repository has earned 231 stars.

[1] `https://commons.apache.org/`
[2] `https://github.com/apache/commons-net`

1.3 Report Evaluation Criteria

This section outlines the main criteria for evaluation of the analyzed project.

The following have been satisfied:

- **Buildability:** Builds are available both locally and through CI/CD. The build process is described in Chapter 2.
- **SonarCloud:** A thorough software analysis has been conducted, and issues have been categorized and fixed through refactoring, in accordance with the selection criteria outlined in Chapter 3.
- **Coverage:** Code coverage has been inspected using JaCoCo: results have been uploaded to Codecov, and they are described in Chapter 4.
- **Mutations:** A mutation testing campaign has been conducted using Pitest. Its results can be found in the pit-report file, and they are described in Chapter 4.
- **Performance:** Performance tests have been created using JMH: their results are visualized with the aid of graphs in Chapter 4.
- **Automation:** Automated tests have been generated with EvoSuite and Randoop. The process is described in Chapter 4.
- **Security:** The project's security has been analyzed with dedicated tools, such as OWASP FindSecBugs and OWASP DC. Their execution reports, found in the files spotbugs-gui and html report, are reported in Chapter 5.

CHAPTER 2

Software Analytics

2.1 Introduction

In order to start analyzing the project, its original repository has been forked[1].
The project was then cloned from the repository onto local installations of IntelliJ IDEA, and using Apache Maven, it has been successfully compiled, tested and packaged.
Local builds were created for two different operating systems, Windows 10 and Arch Linux, both of which were using the latest LTS version (21) of Java Development Kit.
Successful builds using CI/CD with GitHub were also triggered, as a result of the activation of different Actions following commits and merge requests.
The reference platform for the CI/CD pipeline is the latest LTS version of Ubuntu, running JDK 21.

[1] `https://github.com/Tensa53/commons-net`

2.2 SonarCloud Analysis

The project has been analyzed using the SonarCloud[2] service. Found issues are visualized on a dashboard[3] and divided in the following categories:

Bugs: A coding error that will break your code and needs to be fixed immediately. Bugs fall into the *reliability* attribute of software quality. **43** bugs have been found in the source code, divided into three levels of severity:

- **High**: 19
- **Medium**: 23
- **Low**: 1

Code Smells: Code that is confusing and difficult to maintain. Code Smells are related to the *maintainability* attribute of software quality. **1020** code smells have been found:

- **High**: 105
- **Medium**: 513
- **Low**: 402

Fixing the found bugs and code smells is crucial to improve the *consistency*, *intentionality*, *adaptability* attributes and to achieve a cleaner code. Such issues are further analyzed in Chapter 3 - Bug Fixing and Refactoring.

Vulnerabilities: Code that can be exploited by hackers. Vulnerabilities are connected to the *security* attribute of software quality. **2** vulnerabilities have been found:

- **High**: 2
- **Medium**: 0
- **Low**: 0

[2] `https://www.sonarsource.com/products/SonarCloud/`

[3] `https://SonarCloud.io/summary/new_code?id=Tensa53_commons-net`

Security Hotspots: Security-sensitive code that requires manual review to assess whether or not a vulnerability exists. Security Hostpots are linked to the *security* attribute of software quality. **59** security hotspots have been found:

- **High**: 0
- **Medium**: 2
- **Low**: 57

Fixing vulnerabilities and security hotspots is key to improve the *responsibility* attribute and to achieve a cleaner code. Such issues are further analyzed into Chapter 5 - Software Vulnerabilities.

CHAPTER 3

Bug-fixing and Refactoring

3.1 Introduction

After acquiring SonarCloud data on existing bugs and code smells, we have decided to prioritize the refactoring of problems related to the adaptability attribute, in order to achieve a cleaner code.

Source code is considered to be adaptable when it is structured in such a way that it evolves easily and can be developed with confidence. Having adaptable code makes extending or repurposing its parts easy, and it also promotes implementing localized changes without undesirable side-effects. Adaptable code is focused, distinct, modular, and tested.

These characteristics suit the Commons-net project very well, since it is a library that implements many basic Internet protocols. Since new Internet protocols may be built on top of existing protocols, it is vital that the code be very adaptable.

3.2 Bugs

Out of the 43 bugs initially considered within the report provided by SonarCloud, our attention was focused on 19 of them, as they had High Severity, and thus were more likely to cause severe malfunctions if they were not fixed.

We divided these in two categories:

- **Try-with-resources without finally clause: (17)**
- **missing end conditions in loop: (2)**

Both bugs related to missing end conditions in loops were fixed without any particular problems: on the contrary, those related to missing finally clauses presented additional complexities.

Out of the 17 that were reported as belonging to this category, 5 turned out to be false positives: the static analysis performed by SonarCloud fails to account for cases in which resources are returned at the end of a function.

The addition of a finally clause to the try catch block in which these resources were declared would have resulted in the contested resources being closed before being returned to the calling function: the same thing would have happened if the resource was declared inside of the try block of a try-catch-finally statement.

A closed resource is useless to the calling function: therefore, these 5 bugs were marked as false positives on the SonarCloud platform, and we decided not to modify their functions.

In all cases except one, in which the respective functions had to be refactored in order to allow for automatic closing of the resources involved, the remaining 12 bugs were fixed by declaring the resources inside of a newly added try block, as part of a try-catch-finally statement.

3.3 Code Smells

Initially, 105 High Severity Code Smells were identified through SonarCloud analysis. Our efforts were focused on fixing those connected to the Adaptability attribute, of which there were 72.
Plans were made to put additional man-hours towards fixing the remaining 33 High Severity Code Smells: however, it was deemed more advantageous to concentrate solely on the Adaptability attribute, and to instead reserve more hours for the testing phase.

The Code Smells thus identified were divided in 3 categories:

- **Refactoring a method to reduce its cognitive complexity (43);**
- **Defining a constant instead of duplicating a literal (6);**
- **Adding at least one assertion to a test case (23).**

The 23 Code Smells related to missing assertions in test cases were determined to be false positives, as their methods relied on the absence of these assertions, as implemented by their original authors.

Some of these test cases checked whether an exception had occurred, and if not, then the test cases were successful. Therefore, there was no need for an assert instruction, neither to pass nor to fail the test, as it did so automatically.
Others were instead test cases which were inherited from a common implemented interface, and were therefore not applicable to some of the extending classes. Several comments were left to indicate this, such as "Unused" or "Done in other classes".
All Code Smells related to multiples of the same literal were fixed by substituting them with constants.

All but one of the Code Smells related to refactoring complex methods were fixed: most of the hours allocated to Bug-fixing and Refactoring were spent on these, as knowledge of the codebase, underlying classes and methods and control flows was necessary. It should also be noted that throughout the project a variety of coding

styles were used, a problem which became especially pronounced when debugging functions with complex control flows.

In most cases, the functions which needed refactoring were not split into different methods, with each method having unique responsibilities pertaining to a certain task.
Such functions are commonly known as "Brain Methods", composed of many Lines of Code, and characterized by high cyclomatic and cognitive complexity.
Refactoring these methods is simple when most of the variables used are global variables, such as in most cases found.

There were additional problems, however, when the variables used were local to separate scopes, such as when working with threads, synchronized resources, or methods which cannot throw exceptions (for example, run() in a class that implements Runnable). In these cases, the solution chosen was to re-implement the relevant methods starting from scratch, and choosing a different control flow, which uses less instructions such as break or continue.

The only function that was not refactored has, according to SonarCloud analysis, a Cognitive Complexity of 150: for reference, the average CC found for these methods was between 30 and 50. Since it would take a lot of effort to refactor it, we decided not to do so, and to instead advance to the testing phase.

CHAPTER 4

Testing

4.1 Introduction

Commons-net already has a test suite, implemented with JUnit[1]. Following execution of the test suite using Maven, the results show that out of the 415 tests run, none fail or print warnings: however, 10 tests are skipped.

4.2 Coverage Analysis

The project provides support for coverage analysis. JaCoCo[2] results are processed by a Maven plugin through a GitHub Action, and subsequently sent to the Codecov[3] service, which visualizes them on its dashboard[4]. The test suite is able to reach **33.38%** of coverage, with **5/19** packages reaching at least **50%**:

- **ftp** - **55.52%** of coverage;
- **tftp** - **66.43%** of coverage;

[1] `https://junit.org/junit5/`
[2] `https://www.jacoco.org/jacoco/`
[3] `https://about.codecov.io/`
[4] `https://app.codecov.io/gh/Tensa53/commons-net`

- **util** - **66.43%** of coverage;
- **telnet** - **67.60%** of coverage;
- **ntp** - **71.56%** of coverage;

On the other hand, **8/19** packages are not covered at all:

- **discard** - **0.00%** of coverage
- **whois** - **0.00%** of coverage
- **echo** - **0.00%** of coverage
- **pop3** - **0.00%** of coverage
- **finger** - **0.00%** of coverage
- **chargen** - **0.00%** of coverage
- **daytime** - **0.00%** of coverage
- **bsd** - **0.00%** of coverage

4.3 Mutation Testing

Mutation Testing aims to evaluate how well the test suite behaves by changing some portions of the source code exercised by the test cases. With Pitest[5], it is possible to conduct a mutation testing campaign.
The default configuration was used for the mutant operators. The gathered results show that out of **168** classes, the campaign resulted in:

- Line coverage: **37%**;
- Mutation coverage: **30%**;
- Test strength: **78%**.

[5] `https://pitest.org/`

4.4 Performance Testing

Performance Testing was carried out locally on the machine running Windows 10, using Java Microbenchmark Harness to test execution times for single methods and the test suites that they are a part of.
The following test classes were considered for this phase, as their execution times were greater than 2 seconds/operation:

- **TestNtpClient** - time elapsed: **2.009 s/op**
- **TFTPTest** - time elapsed: **18.560 s/op**
- **FTPSClientTest** - time elapsed: **6.794 s/op**
- **TelnetClientTest** - time elapsed: **47.03 s/op**

After applying various optimizations, which are detailed in the following lines, the test classes' execution times changed to:

- **TestNtpClient** - time elapsed: **0.111 s/op**
- **TFTPTest** - time elapsed: **18.560 s/op**
- **FTPSClientTest** - time elapsed: **6.794 s/op**
- **TelnetClientTest** - time elapsed: **6.906 s/op**

The FTPSClientTest class has not been analyzed in depth due to it lacking a default public constructor, and therefore being incompatible with JMH's @State annotation, which is required to run benchmarking tests.
With the exception of the latter, the impact of individual methods on test performance was analyzed and visualized with Python's MatPlotLib library.
The following subsections detail the results obtained after applying eventual optimizations:

4.4.1 TestNtpClient

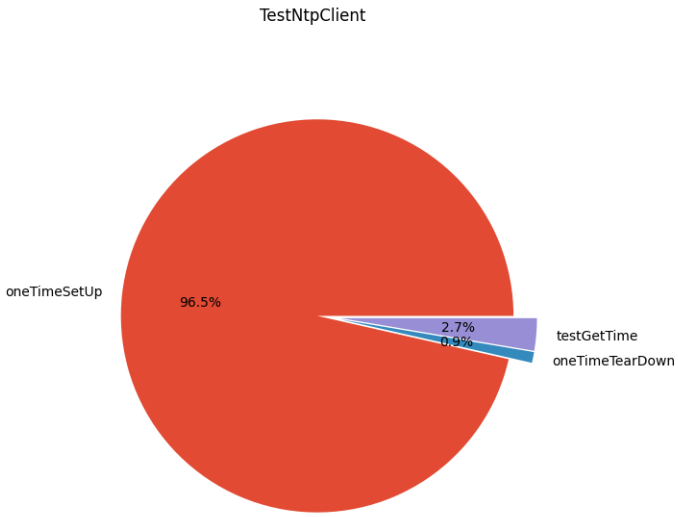

Figure 4.1: Pie Chart times for TestNtpClient methods

In this class, the oneTimeSetUp method takes the longest time to execute, as it is responsible for creating and starting an NTP server. It tries multiple times to confirm that the server has been started using Thread.sleep(1000).
Runtime was drastically reduced by setting the duration parameter of the sleep invocation to 100: a value of 50 was also tested, however it led to failure of connected test cases.

4.4.2 TFTPTest

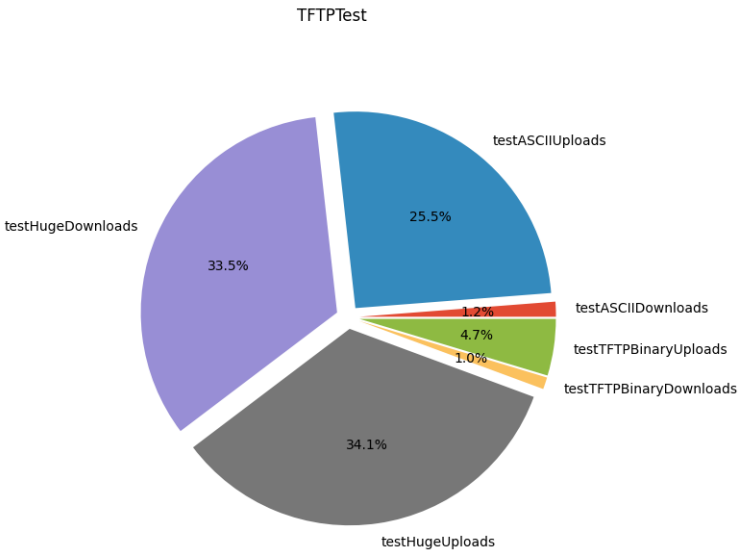

Figure 4.2: Pie Chart times for TFTPTest methods

These methods' times are consistent with their functionalities: operations on binary files take less time than the others, while operations on files defined as "Huge," which weigh between 1 and 37 GB, take the longest time to complete.
The following graph illustrates the relationship between the total file sizes used for download and upload methods and their runtime.

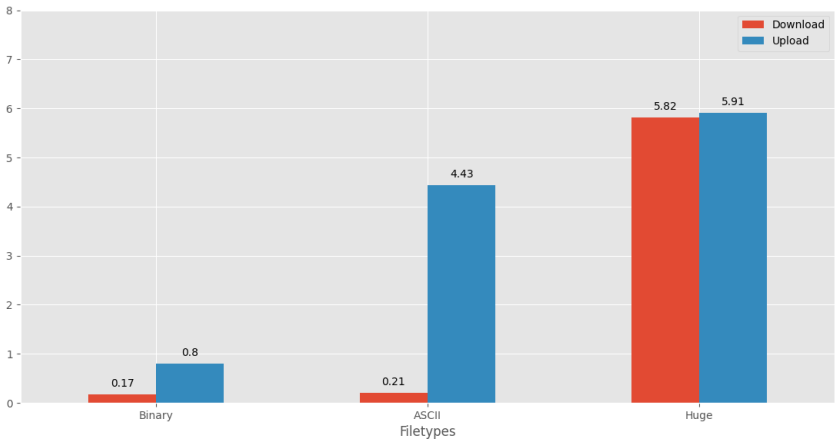

Figure 4.3: Bar Graph relation between methods in TFTPTest

This graph denotes as outliers the ASCII methods, since testASCIIUploads is much slower than testASCIIDownloads, especially when contrasted with the same difference found in the other methods.

A crucial difference has been found in the invocation of the sendFile method with the *ASCII_MODE* parameter, which results in the usage of a ToNetASCIIInputStream, rather than a regular FileInputStream. Operations with the former are far slower because of the different implementation of the read() method.

By removing the use of ToNetASCIIInputStream, it is possible to reduce the execution time of the testASCIIUploads method to 0.794 s/op: however, this alters the executed instructions significantly, in a way that results in loss of functionality, even if the test suite does not detect any errors. Therefore, the method was not modified.

4.4.3 TelnetClientTest

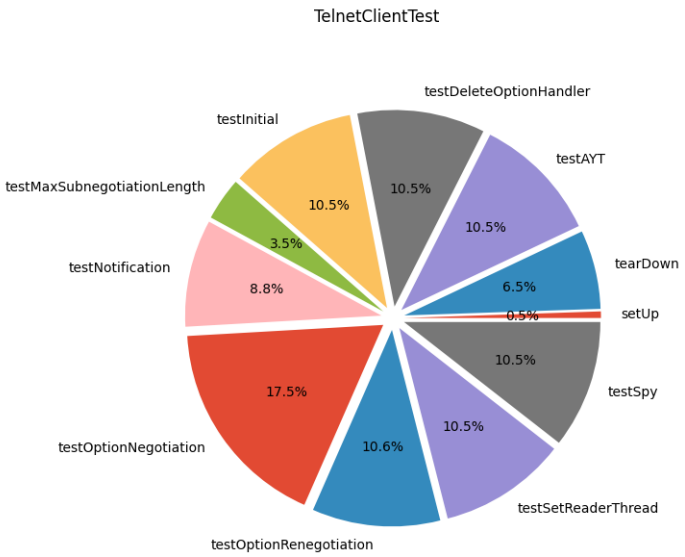

Figure 4.4: Pie Chart times for TelnetClientTest methods

Execution times of this class's methods are consistent with their purpose, as they test similar functionality and have similar computing loads, relative to instructions executed and the scopes of their variables.

However, it was noted that all of the examined functions use Thread.sleep(). The provided parameters were 1000 when certain operations were previously made with OutputStream, and 500 for operations with InputStream. By reducing these values to 100 and 50 respectively, a significant increase in execution speed was observed, without resulting in failure of the test suite.

4.5 Improving the coverage

Considering the low percentage of coverage achieved, it was possible to use several tools to automatically generate new test cases and improve this percentage.
Since many test cases were generated, which might need a specific runtime or tools to be executed, we decided to not add it to the original test suite of the project: instead, we only analyzed the possible impact of adding them.

We considered the packages that haven't been adequately tested, in particular the ones where reported coverage was 0%.
In this way, without adding the generated test cases inside the project and without calculating once more the coverage percentages, we were still able to notice an improvement in how much they benefited from the new test cases.

4.5.1 EvoSuite

EvoSuite[6] was able to generate test cases for every class of the 8 analyzed packages, except for POP3Reply. This tool was also able to write the produced results to a CSV file, noted as a table on the next page.
Any further information about the report's type is found in the EvoSuite documentation.[7]

[6] `https://www.evosuite.org`
[7] `https://www.evosuite.org/documentation/tutorial-part-3/`

Target_Class	Criterion	Coverage	Total_Goals	Covered_Goals
org.apache.commons.net.discard.DiscardTCPClient	DEFAULT	0.93755	19	18
org.apache.commons.net.discard.DiscardUDPClient	DEFAULT	0.94444	63	59
org.apache.commons.net.whois.WhoisClient	DEFAULT	0.72916	37	28
org.apache.commons.net.echo.EchoTCPClient	DEFAULT	0.93755	19	18
org.apache.commons.net.echo.EchoUDPClient	DEFAULT	0.89583	72	62
org.apache.commons.net.pop3.POP3Reply	DEFAULT	0.00000	2	0
org.apache.commons.net.pop3.ExtendedPOP3Client	DEFAULT	0.19725	184	20
org.apache.commons.net.pop3.POP3MessageInfo	DEFAULT	0.91666	54	52
org.apache.commons.net.pop3.POP3	DEFAULT	0.60632	417	211
org.apache.commons.net.pop3.POP3Client	DEFAULT	0.59302	791	458
org.apache.commons.net.pop3.POP3SClient	DEFAULT	0.57749	302	155
org.apache.commons.net.pop3.POP3Command	DEFAULT	0.77083	29	24
org.apache.commons.net.finger.FingerClient	DEFAULT	0.46499	155	89
org.apache.commons.net.chargen.CharGenUDPClient	DEFAULT	0.89583	71	65
org.apache.commons.net.chargen.CharGenTCPClient	DEFAULT	0.93755	19	18
org.apache.commons.net.daytime.DaytimeUDPClient	DEFAULT	0.89583	57	50
org.apache.commons.net.daytime.DaytimeTCPClient	DEFAULT	0.39853	47	22
org.apache.commons.net.bsd.RCommandClient	DEFAULT	0.52397	411	302
org.apache.commons.net.bsd.RExecClient	DEFAULT	0.52430	224	95
org.apache.commons.net.bsd.RLoginClient	DEFAULT	0.81845	55	47

Table 4.1: The report generated by EvoSuite

4.5.2 Randoop

Since we were already satisfied by the results obtained with EvoSuite, we decided to not take into consideration the test cases generated by Randoop[8]: nonetheless, this tool was also used to find additional interesting information.

Considering the 8 packages with no coverage, Randoop was executed in an infinite loop with the 3 packages named echo, chargen and daytime. For the other 5 packages, Randoop was able to generate test cases: however, the package named discard generated some flaky tests, along with notifications about methods, which may have been responsible for the flakiness.

[8] `https://randoop.github.io/randoop/`

The methods are listed as follows:

- org.apache.commons.net.DatagramSocketClient.getLocalPort()
- org.apache.commons.net.DatagramSocketClient.getDefaultTimeout()
- org.apache.commons.net.SocketClient.connect(java.net.InetAddress,int)
- org.apache.commons.net.DatagramSocketClient.getSoTimeoutDuration()
- org.apache.commons.net.DatagramSocketClient.getSoTimeout()
- org.apache.commons.net.DatagramSocketClient.open(int)
- org.apache.commons.net.discard.DiscardUDPClient.send(byte[],int,java.net.InetAddress)
- org.apache.commons.net.DatagramSocketClient.setCharset(java.nio.charset.Charset)
- org.apache.commons.net.DatagramSocketClient.setSoTimeout(int)
- org.apache.commons.net.DatagramSocketClient.getCharset()

CHAPTER 5

Software Vulnerabilities

5.1 Introduction

A vulnerability scan was already conducted with SonarCloud, as stated in Section 2.2 - SonarCloud Analysis. FindSecBugs[1] and Dependency-Check[2] were also executed in order to detect vulnerabilities matching different heuristics inside of our source code and related dependencies.

5.2 SonarCloud

SonarCloud was able to find 2 software vulnerabilities with high severity and 2 security hotspots with medium severity, all of which were connected to the Responsibility attribute. Our aim was to fix each of these issues. The vulnerabilities are related to server certificate validation on an SSL/TLS connection, while the security hotspots are respectively related to Denial of Service and Weak Cryptography.

[1] https://find-sec-bugs.github.io/

[2] https://owasp.org/www-project-dependency-check/

5.3 FindSecBugs

A FindSecBugs campaign was already conducted by the authors of the original repository. Some vulnerabilities were classified as false positives and were therefore included in a filter list, which resulted in FindSecBugs ignoring them.
We decided to trust the original authors' judgement, and not to analyze these vulnerabilities ourselves, but instead only those not present in the filters.
The execution of FindSecBugs with the filter deactivated resulted in the following list:

- Malicious code vulnerabilities:
  - Method returning array may expose internal representation:
    - **May expose internal representation by returning reference to mutable object (14)**
  - Mutable static field:
    - **Public static method may expose internal representation by returning array (1)**
  - Storing reference to mutable object:
    - **May expose internal representation by incorporating reference to mutable object (16)**

Running FindSecBugs with the filter disabled resulted instead in:

- Malicious Code vulnerability:
  - Mutable static field:
    - **Public static method may expose internal representation by returning array (1)**

5.4 Dependency-Check

A Depedency-Check campaign was conducted and resulted in no vulnerable dependencies being found.
Commons-net only has two dependencies related to the slf4j[3] library, SLF4J API Module[4] and SLF4J Simple Provider[5] , neither of which have existing vulnerabilities.

5.5 Fixing the Vulnerabilities

The first vulnerabilities to be considered were those found through SonarCloud, two of which were related to the validation of SSL/TLS certificates.
In one instance, the related class, FTPSTrustManager was deprecated, while in the other case, server-side certificates were correctly validated: the error was thrown in response to a lack of validation of client-side certificates. However, as this class was found in numerous other open source projects without any validation of client-side certificates, and the correct example provided by SonarCloud's static analysis also lacked such validation, this vulnerability was assessed as a false positive.

Regarding the medium severity security hotspots, both were examined and determined to not be security risks, according to the criteria provided by SonarCloud in its "Assess" tab.
The first item of interest was a possible denial of service through the analysis of a regex expression: since said expression is hardcoded, and the function using it is pattern.compile(), which is not vulnerable to polynomial runtime due to backtracking and does not have any defined timeouts, there is no security risk.

The other hotspot was the usage of Random.nextInt() to generate a random port number, on which an FTP client could be started. Since no encryption was used in this case, and there is no need for pseudo-randomness, no security risks were found.

[3] https://www.slf4j.org/
[4] https://mvnrepository.com/artifact/org.slf4j/slf4j-api/2.0.9
[5] https://mvnrepository.com/artifact/org.slf4j/slf4j-simple

Afterwards, the malicious mutable static field spotted by FindSecBugs was considered: the corresponding vulnerability was classified as *MS_EXPOSE_REP*, in which a public static method returns a reference to a mutable object or an array that is part of the static state of the class.

Any code that calls this method can freely modify the underlying array.
In this case, the method getInstance() returned a reference to the MSLxEntryParser class, which contained three static arrays. However, since these arrays were also declared as private and final, none of them could be accessed from said reference.
In previous versions of FindSecBugs, this vulnerability was often falsely identified: therefore, we considered it a false positive.

CHAPTER 6

Conclusions

All requirements for a successful analysis of the Apache Commons Net library's dependability attributes have been satisfied, including the tasks outlined in each chapter of this report, as well as all evaluation criteria being met.
Various metrics were used to convey the effects of this analysis on the project: the following table outlines them.

SonarCloud Analysis (Software Analytics)		
Bugs		
Severity		
High	Medium	Low
19	23	1
Code Smells		
Severity		
High	Medium	Low
105	513	402

Continued on next page

Vulnerabilities & Security Hotspots		
Severity		
High	Medium	Low
2	2	59
Codecov Analysis (Coverage computation)		
Code coverage		33,38%
Pitest Campaign (Mutation Testing)		
Line coverage		37%
Mutation coverage		30%
Test strength		78%
Java Microbenchmark Harness (Performance Testing)		
Execution times (before)		
Class		Time elapsed
TestNtpClient		2.009 s/op
TFTPTest		18.560 s/op
FTPSClientTest		6.794 s/op
TelnetClientTest		47.03 s/op
Execution times (after)		
Class		Time elapsed
TestNtpClient		2.009 s/op
TFTPTest		18.560 s/op
FTPSClientTest		6.794 s/op
TelnetClientTest		47.03 s/op

Continued on next page

EvoSuite Campaign (Generation of test cases)		
Package	Total goals	Covered goals
discard	82	77
whois	37	28
echo	91	80
pop3	1779	920
finger	155	89
chargen	90	83
daytime	104	72
bsd	690	444
Randoop Campaign (Generation of test cases)		
Package	Successful	Flakiness
discard	YES	YES
whois	YES	NO
echo	NO	NO
pop3	YES	NO
finger	YES	NO
chargen	NO	NO
daytime	NO	NO
bsd	YES	NO
FindSecBugs Scan (Project Security)		
Without filter list (true and false positives)		
Malicious code vulnerabilities		31
With filter list (true positives)		
Malicious code vulnerabilities		1

Table 6.1: Recap table.